RL-TR-93-60
Final Technical Report
May 1993

INCREMENTAL REDERIVATION OF SOFTWARE ARTIFACTS: FY 92

The MITRE Corporation

Melissa Chase, Howard Reubenstein, and Alexander Yeh

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-93-60 has been reviewed and is approved for publication.

APPROVED:

DOUGLAS A. WHITE
Project Engineer

FOR THE COMMANDER:

JOHN A. GRANIERO
Chief Scientist
Command, Control and Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CA) Hanscom AFB MA 01731. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1993	Final Oct 91 - Sep 92

4. TITLE AND SUBTITLE
INCREMENTAL REDERIVATION OF SOFTWARE ARTIFACTS: FY 92

5. FUNDING NUMBERS
C - 19628-89-C-0001
PE - 62702F
PR - 5581
TA - 27
WU - 57

6. AUTHOR(S)
Melissa Chase, Howard Reubenstein and Alexander Yeh

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The MITRE Corp
202 Burlington Road
Bedford MA 01730

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CA)
525 Brooks Road
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-93-60

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Douglas A. White/C3CA/(315)330-3564.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

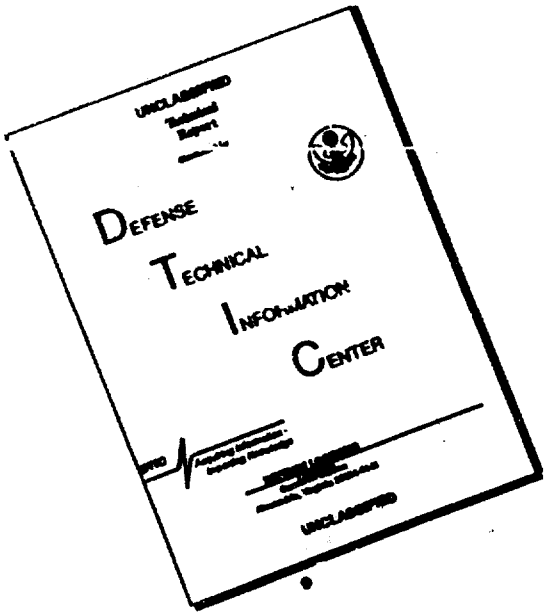
13. ABSTRACT (Maximum 200 words)

Design relay presents a possible enabling technology to the Knowledge-Based Software Assistant specification maintenance and implementation derivation approach to software development. More generally, design replay can also be applied to derivations between a variety of different software description abstraction levels. Radical changes to a software artifact cannot generally be addressed by designed replay as they require new design input. Evolutionary changes are more amenable to design replay and often involve incremental changes to derived artifacts. This report outlines an approach to rederivation that exploits the incrementality of evolutionary maintenance changes, describes the state at which the implementation of this approach is at, and what remains to be done to test this approach.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Knowledge-Based Software Engineering, Software Engineering, Automatic Programming, Formal Specifications, Artificial Intelligence, Evolutionary Development, Program Derivation, Replay, Rederivation	24
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	U/L

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Incremental Rederivation of Software Artifacts: FY92 Final Report

Melissa P. Chase, Howard Reubenstein and Alexander Yeh
The MITRE Corporation
202 Burlington Road
Bedford, MA 01730
[pc,hbr,asy]@mitre.org

January 20, 1993

Abstract

Design replay presents a possible enabling technology to the Knowledge-Based Software Assistant specification maintenance and implementation rederivation approach to software development. More generally, design replay can also be applied to derivations between a variety of different software description abstraction levels. Radical changes to a software artifact cannot generally be addressed by design replay as they require new design input. Evolutionary changes are more amenable to design replay and often involve incremental changes to derived artifacts. This report outlines an approach to rederivation that exploits the incrementality of evolutionary maintenance changes, describes the state at which the implementation of this approach is at, and what remains to be done to test this approach.

1 Introduction

The problem we are exploring is the insertion of a design replay capability into Rome Lab's Knowledge-Based Software Assistant (KBSA) [1]. The KBSA is composed of a number of facets that roughly correspond to activities in a standard waterfall life-cycle model. These facets include requirements, specification, and development. Note that the KBSA does not force a waterfall development, in fact it is based on a new paradigm of software development involving maintenance of specifications and rederivation of implementations. Each facet assists in a design activity producing increasingly formal descriptions of a desired system. Each facet can be viewed as taking a transformational design approach.

Our task is constrained by two important requirements. First, we wish to provide a design replay utility that can be used by any and all of the facets. Thus, we are likely to take an approach reminiscent of the Joshua system's [2] "protocol of inference." We are interested in designing a protocol that a design tool can use to record the: decisions it is making, alternatives it is considering, goal structure of its problem solving, and data dependencies

between actions it takes and artifacts created. This protocol should permit capture of the information necessary to permit application of incremental redesign techniques.

Second, we want to be able use incremental replay from the *current* state of the code artifact to reconstruct the code for a modified design. (In this context, the use of the word code should be qualified as referring to the output artifact of a particular facet. This artifact may be source code, but it could be specification, requirements or other descriptions.) We do not want a strategy that requires replay of an entire design.

To explain an important difference between software design problems and other areas amenable to replay technology, planning for example, particularly with respect to this second requirement, an analogy to the blocks world is helpful. In blocks world planning there are two artifacts of particular concern: the plan to achieve the desired goal and the actual configuration of the blocks (before and) after executing the plan. In software, the corresponding artifacts are the design (plan) to produce the code and the code itself (block configuration). Incremental replanning techniques [3] assist in the creation of a new plan/design, but not in the formation of a new block configuration/code from the state left by the original plan. This latter capability is critical in software construction where the design has already been applied to yield code.

Maintenance must proceed from this existing code for a number of reasons. The design process is likely to have been quite long, e.g., 10,000 transformations [4] for a medium-sized problem, and on average might only be 90% automated. Therefore, replay of an entire design would require user assistance in remaking 10% of the design decisions, e.g., 1,000 decisions, many of which may not even have been affected. Even if the process were 99% automated, 100 decisions would still have to be remade by the user. Also, depending on the efficiency of the transformation system, even fully automatic replay of 10,000 transformations may be something to avoid. In essence, there are two incremental modification problems to be considered simultaneously in software design, that of modifying the design and that of modifying the source code.

The next section describes our approach to design replay. Following this are sections describing the state of our implementation and related work.

2 Incremental Rederivation

A major complication in design replay is the so called correspondence problem [5, 6]: establishing a correspondence between the components of the original initial specification and the changed initial specification. This problem is both difficult and necessary to solve for replay to occur: the original set of transformations need to be replayed on the components of the changed initial specification that correspond to the components of the original initial specification which the set of transformations were originally applied to. An interesting aspect of the incremental rederivation approach is that it finesses this correspondence problem to a large extent. The input to the replay capability consists of: an original initial specification, a transformational development and its associated dependency information as described below, a solution artifact, and a delta to the original specification. The required output is a new solution artifact that reuses as much of the initial solution as possible and minimizes its interaction with the underlying design performance system/agent. Correspondence is not a

significant issue under this problem definition because the change is localized to an area of the original specification and the dependency structure identifies and propagates the other correspondences in the transformational development.

To explore this further, we need to look at an initial simple definition for transformations and a basic artifact representation. The transformation representation we are concerned with is derivative from [7] and is meant to capture only the simplest syntactic properties of a transformation. Similarly, the artifact representation, which must be facet independent to a large extent, captures only basic syntactic properties.

Artifacts will be represented as abstract syntax trees (AST). This provides a level of representation above the purely textual but without facet specific semantic concepts.[1] Transformations, at this initial level of description, consist of an input pattern that matches pieces of the base AST and an output modification to the AST. The section of the AST to be modified must be included in the input pattern. The parts of the AST matched by the input pattern are called the *input span* of the transformation and the modified sections comprise the *output span*.

2.1 Macro-Level Rederivation

Given an input specification $S0$ consisting of disjoint sub-components (S_1, S_2 ... S_n), and a solution artifact Sf consisting of disjoint sub-components (Sf_1, Sf_2 ... Sf_n), and a transformational development history with input and output span information preserved, changes to the input specification can be propagated at a macro-level as follows: given[2] a change to a particular S_i we can compute a partition of Sf into two sets of subcomponents, those that are definitely unchanged and can therefore be reused intact and those that are possibly changed.

This macro-level rederivation is a form of impact analysis that in itself can be useful in large systems under the assumption that many of the maintenance changes made to a specification only have relatively local effects in the solution artifact. This assumption is restrictive but consistent with the typical maintenance profile where the difficult problem is reliably identifying the relatively small percentage of the code that needs to be changed to respond to a maintenance request.

Given the set of possibly changed Sf_i, we can attempt to replay the transformational history to rederive new solution sub-components. For each transformation that we might consider replaying however, there is a spectrum of possibilities from redo (and possible reinvocation of the design system/agent) to having extracted finer-grained transformation dependencies that permit more precise control of reinvocation of a transformation or dependency-directed reinvocation of that transformation. This finer level of dependency information is discussed in the next section.

Two goals in keeping this dependency information should be recalled here. First, we have

[1] Facet specific semantic concepts can be helpful in rederivations but are excluded for now because their use will vary from one facet and system to another.

[2] These various inputs are provided by a design tool interacting with the replay component through the defined information and dependency protocol. This protocol defines the data a design tool must record about its ongoing design process in order to make use of the replay facilities. The protocol will not be discussed further in this paper.

the goal of essentially performing impact analysis through the dependency structure. This analysis gains us time efficiency by both not replaying entire huge derivations (which is traded off in space required for the dependencies) and not interacting with a person for manual information extraction. Second, we wish to be able to closely analyze those transformations that are possibly affected by new input requirements. Most transformations, in general, will be unaffected and therefore not require replay. Those that are affected and still applicable will often be modifiable through the dependency structure. If a transformation is no longer applicable and the design program needs to be reinvoked, then it is likely that rederivation of the entire subsolution will be necessary. If an appropriate impact analysis is supported, then this will be more efficient as fewer valid solution components will be needlessly rederived.

2.2 Micro-Level Rederivation

Before discussing finer-grained dependency information, we need to refine our representation of transformations. We have stated that transformations consist of an input pattern and an output modification. Following Balzer in [7], a transformation also consists of a set of local bindings (used to compute partial substructures) and a set of constraints to be verified. All data accessed in the constraints, bindings, or output modification must be accessed by the input pattern, i.e., the input span of a transformation must identify all data dependencies.

An input pattern implicitly embodies two kinds of constraints, i.e., equality to a constant and equality between substructures (use of the same named pattern variable).

Bindings can perform arbitrary computation but we make a few assumptions about them. First, we assume all data accessed by the binding computation is functionally indicated in the binding calculation call. This implies that a binding calculation is time invariant, i.e., a calculation always returns the same result on the same arguments and does not access internal state. Second, we assume none of the bindings are dead/unused. We could attempt to compute this and, in fact, dead bindings should not affect the ultimate propagation through the dependency structure.

Constraints may also perform arbitrary computation, perhaps accessing a reasoning system to check properties of the input pattern. As with bindings, we assume all these computations are also functional.

Given this finer-grained transformation representation, it is now possible to either compute the fact that a delta to a transformation's input pattern actually has no effect on the validity of the output modification or, more likely, that the output modification can be updated via an incremental recomputation not requiring reinvocation of the transformation. To understand how this will work, we must look at the kinds of changes that can occur in the input span of a transformation and how they propagate through the transformation.

2.3 Replay Delta Calculus

To enumerate the space of deltas that can affect a transformation, we shall step through the computation of whether a transformation is still applicable (see figure 1). A delta to a piece of an AST is postulated and the macro-level dependency mechanism identifies a transformation as having an input span that is possibly affected by the delta. First, the

input pattern must be matched against the changed AST and the delta localized to the affected portions of the input pattern.[3]

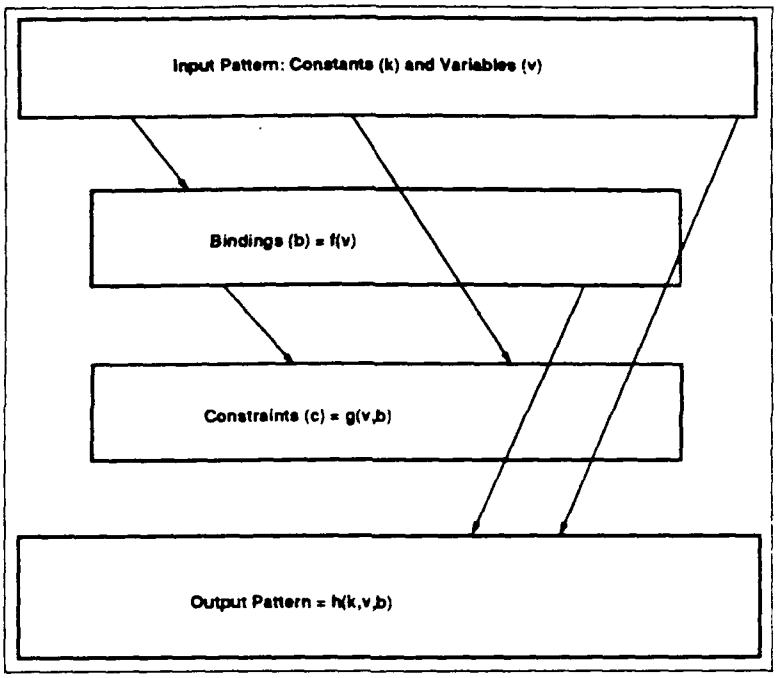


Figure 1: Propagating a Delta Through a Transformation

If the change in the input pattern is in a constant portion of the pattern, then the transformation is no longer applicable. If the transformation is to a variable portion of the pattern then we continue examining the change. Next, we compute a delta set on the bindings from the delta set of the input pattern. Then we compute a possibly delta set for the constraints (and any equality constraints implicit in the input pattern) based on the delta bindings and variables. For each possibly changed constraint, we reevaluate its truth. If any constraint is false, the transformation is no longer applicable. If all the constraints are still true, we move on to computation of the new output pattern.

The output pattern of a transformation may not be changed if all deltas simply affect triggering conditions. The output pattern is changed if it incorporates any of the changed inputs or bindings. At this point, we could simply recompute the new output pattern by recomputing the output modification of the transformation. However, it is also possible to propagate the deltas through the output pattern instead of performing a full recompute. This has the advantage of not requiring reinvocation of the transformation and permitting localization of the deltas to parts of the output pattern.

The possible actions in an output pattern include: creating a constant subcomponent (this is unaffected by deltas in the input), copying a variable in as a subcomponent, or copying a binding in as a subcomponent.

[3]The original match of the input pattern to the AST could be kept as part of the dependency structure and then a delta to a piece of the AST could be mapped directly to the affected part of the input pattern.

The possible deltas that affect the output pattern include:

1. A change in an input constant component.
2. A change that occurs in a single output subcomponent that is copied from a once occurring input variable or binding.
3. A change that occurs in a single output subcomponent that is copied from a multiply occurring input variable.
4. A change that occurs in multiple output subcomponents that are copied from a once occurring input variable or binding.
5. A change that occurs in multiple output subcomponents that is copied from a multiply occurring input variable.

A type 1 change will (as discussed) render a transformation inapplicable. In the other change types, if an output subcomponent is tested in some way (for example, asking if it is of a certain type or value), the transformation may be rendered inapplicable.

In the case where a change occurs to a pattern that appears multiple times in the input, it may be the case that there is an unintentional violation of an implicit program design equality constraint between the multiply occurring subcomponents. Before continuing with replay, we may want to consult the user to verify the intentions.

In the case where a change occurs in multiple output subcomponents versus a single subcomponent, we have a measure of the expanding versus local impact of change.

The output deltas identified from one stage of propagation now form the input deltas for the next stage of propagation which continues until all affected transformations are processed.

2.4 A Simplification Example

The best example of how these techniques work would involve a large derivation and a typical maintenance change affecting only a small part of the implementation artifact. For example, in a database access system consisting of data storage and retrieval routines, data indexes, report generators, and other functions, a user might make a change to an output specification that simply resulted in the alteration of a few constants in the report generation routines. Due to space limitations, we will consider a smaller self-contained example that illustrates how incremental rederivation works.

Suppose we view the development of a small program to compute a formula as a transformational development in which the main activities are the application of simplification rules. Starting with an input formula specification of

$$(P/P * X)/(P + 0)$$

the derivation proceeds through the following stages using the obvious simplification transforms: $(1 * X)/(P + 0)$, $X/(P + 0)$, X/P. Suppose now that a renaming of variables, a typical kind of maintenance change, is proposed as an input delta, i.e., change P to Q.

The dependency structure facilitates this incremental rederivation as follows. The transformation that simplified P/P would be identified by the macro-level propagation as possibly

affected. The micro-level analysis would, however, show the transformation to still be applicable and would further show the output as not dependent on the input variable (i.e., X/X equals 1 for all non zero X, assumed in this case). Propagation of this part of the input change would stop. The transformation that simplified $P + 0$ would also be identified by the macro-level propagation as possibly affected. Still applicable, this transformation's output modification would itself be modified substituting Q for P in the final output.

Instead of rerunning the three transformations, incremental rederivation permits pinpoint updating of only those affected areas of the program. For such a small derivation history, the gains are not significant. For larger derivation histories and those where individual transformations are time consuming, this incremental strategy minimizes the amount of work required to obtain a new artifact.

2.5 The Dependency Structure

Each transformation in a derivation sequence causes a delta to part of the software artifact under construction. The AST for an artifact consists of a set of attributes and keywords organized according to the syntax of the relevant artifact source language.

In recording the transformational development, two views of the artifact are maintained: the original base artifact and the current transformed version. The current version is simply a caching of the results of tracing the transformational development from the original artifact through to the current time. The derivation history is captured as a set of extra-linguistic annotations on the AST rooted initially at the original AST. These annotations capture transformation applications as a tuple of [input- pan, output-span, transform, time-stamp] (and necessary cross-references). The input-span and output-span are identified as sub-trees in the AST. The derivation history annotations include copies of the output-span results of each transformation. The current transformation dependency structure can be traversed by starting at the original base artifact and following the transformation annotations through the structure as ordered by the time-stamp attribute.

Storing this dependency structure will take an amount of memory proportional to how large the input and output components of individual transformations are and the actual number of applied transformations. It is clear that this structure can grow quite large. It can also stay of manageable size, e.g., imagine a 10,000 line program and a 10,000 transformation derivation each of which touches on average 1/10% of the code, 10 lines. This will result in an overhead proportional to 10 times program size.[4] It is clear, however, that the dependency structure will continue to grow as the design process proceeds and thus strategies for compressing this structure must be considered. Use of these strategies will be driven by experiments regarding the actual memory overhead.

A drastic compression strategy would involve discarding the current start state S0 and creating a new start state from some intermediate point in the derivation. Essentially this would discard the ability to roll back a derivation to the original input artifact state. A more moderate strategy would involve keeping the dependency trail for macro-level rederivation, but eliding information necessary for micro-level recomputation. A compromise strategy would involve computing the transitive closure of a sequence of transformations and thus

[4] This is the same size as the estimate given by Neighbors for DRACO[8].

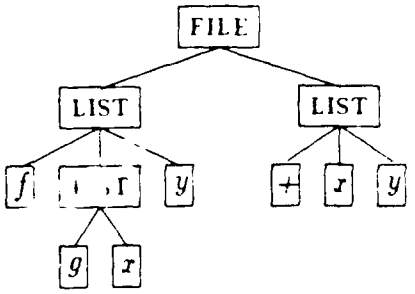


Figure 2: The AST for *(f (g x) y) (+ x y)*

compress the information about that sequence into a single virtual transformation.

3 Implementation

So far, our implementation efforts have concentrated on macro-level rederivation: when an input specification is altered, see which parts of a transformational derivation are definitely unchanged and which parts may be changed. At present, the implementation replays any transformation whose applicability depends on a part that may have changed.

This section of the report describes the following aspects of the current implementation:

1. background,
2. recording the derivation history,
3. marking which parts of the history might be altered when changing part of the original artifact,
4. finding the version of the artifact that is produced as a result of all the unaffected transforms,
5. finding and replaying the possibly affected transforms.

3.1 Background

The implementation is written in REFINE [16]. At present, we have a grammar for a simple Lisp-like syntax.[5] As a result, an initial artifact (specification) needs to be given using a Lisp-like syntax. REFINE will use the grammar to parse an initial artifact to form an *abstract syntax tree* (AST) for that artifact. This AST and its variants are what the rest of the implementation depends on for recording what is going on: in other words, an AST is assumed to be the common level of representation.

An example of an AST is that the lisp file (with 2 top level lists) "*(f (g x) y) (+ x y)*" will get parsed into the AST shown in figure 2.

[5]The only composite objects allowed are files of lisp objects, lists of lisp objects, and a quoted lisp object. The implementation described in this report does not deal with quoted lisp objects.

A derivation is assumed to be a sequence of transformations applied to the initial AST. A transformation can be anything that can be applied to a list of AST nodes. We have been using REFINE rules (named, parameterized transforms). The implementation's requirements for a transformation are as follows: It should take 1 or more arguments. The first argument will be the AST node that may be changed by the transformation.[6] The other (if any) arguments are AST nodes that will *not* be modified by the transformation, but can be used to provide information to determine if the first node should be modified and/or how to modify that node. A final requirement is that if the transformation modifies the first argument, the transformation needs to set the global variable **xformed-node** to point to the result of the modification.[7]

An example transformation is the rule

```
rule pos-abs (abs-expr: user-object, pos-expr: user-object)
  abs-expr = '(abs @x)' & pos-expr = '(≥ @y 0)' & term-equal?(x,y)
           --> (replace abs-expr by x) & (*xformed-node* <- x)
```

This rule simplifies the expressions that have form of the absolute value of some sub-expression to just that sub-expression when that sub-expression is declared to be non-negative. The rule's name is *pos-abs*. The AST node it may modify is called *abs-expr* in the rule, and the other AST node it examines is called *pos-expr*. The rule states that if

1. *abs-expr* corresponds to an expression of the form *(abs α)*, where α is any expression
2. *pos-expr* corresponds to an expression of the form *(≥ β 0)*, where β is any expression, and
3. α and β are equivalent expressions (have ASTs that "look the same")

then

1. replace *abs-expr* with the node corresponding to α ("simplify" *abs-expr*), and
2. make the variable **xformed-node** point to that node also

3.2 Recording the Derivation History

The history is started by adding attributes to the initial artifact's AST and making a copy of that AST. One copy serves as a description of the current artifact. The other copy will record the history of derivations. The nodes in the 2 copies have attributes that point to the corresponding node in the other copy.

To keep the history current, all transformations are carried out via the procedure *apply-rule-to-node*. This procedure takes in the name of the transformation and the arguments for that transformation. The arguments should be nodes from the AST that is serving as a

[6] For now, if one wants to transform more than 1 node at a time, one can either use as the first argument the nearest node that is a common ancestor to all the nodes to be altered, or write a group of transformations that each alter 1 of the nodes to be altered.

[7] This is trickier than it sounds. Sometimes a node is modified by modifying some of its attributes. But sometimes a node is "modified" by replacing the current object representing the node with a new object.

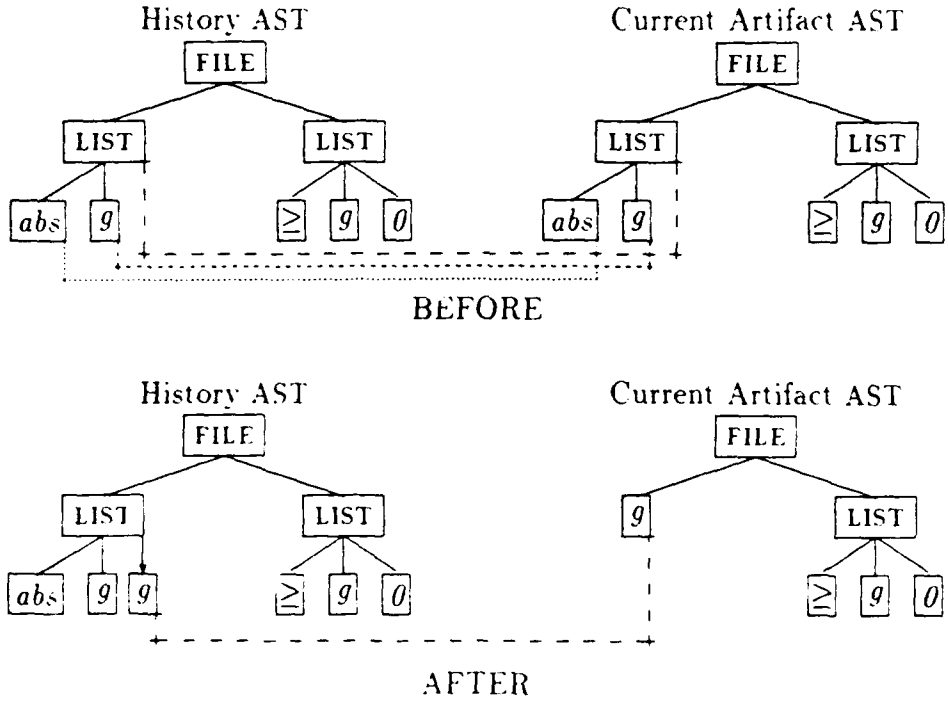


Figure 3: AST's before and after the *abs-expr* transformation of *(abs g)*

description of the current artifact. *apply-rule-to-node* will apply the transformation to those arguments (possibly modifying/updating the current artifact AST in the process) and then update the AST that is recording the history.

A major difference between the AST describing the current artifact and the AST recording the history is in how *apply-rule-to-node* updates them. The node being transformed in the current artifact AST is either modified or replaced to reflect the results of that transformation. The corresponding node in the history AST is neither modified nor replaced. Instead, after transforming that current artifact AST node, a copy of the updated (transformed) node is made (call the copy X) and attached to that corresponding node in the history AST by making X the value of the *xformed-into* attribute of the history AST node. Then the updated current artifact node's corresponding history node is updated to be X.

Figure 3 shows what happens to the 2 AST's when applying the *abs-expr* transform (shown in subsection 3.1) to the node for the expression/list *(abs g)* if the original artifact is a file with the 2 lists *(abs g)* and *(≥ g 0)*. With the presence of the expression *(≥ g 0)*, the transformation converts *(abs g)* into *g*. The solid lines without arrowheads show the sub-expression/parent-expression relationships among the nodes. The solid line with an arrowhead indicates that *(abs g)*'s *xformed-into* attribute points to a *g* node. The dashed lines link the ASTs' corresponding nodes for the expression *(abs g)* both before and after the transformation. To prevent clutter, the links for the ASTs' other corresponding nodes are not shown.

When a node in the current artifact AST is transformed, not only does the corresponding history node store the results of the transformation, that latter node also stores the name of the transformation, how many transformations have been performed so far on the current

artifact AST, and what *supporting* nodes might have been used by the transformation to transform the node the way it did.[8] An example of a *supporting* node is the node for the expression *(≥ g 0)* in each of the ASTs: that node is needed by the *pos-abs* transformation (as the second argument to the transformation) to tell the transformation that *g* is non-negative, so it is okay to simplify *(abs g)* into *g*.

3.3 Marking History Possibly Altered by Changes

After deriving the final artifact, one will often want to change one or more parts of the original artifact (specification). Then one needs to figure out which parts of the derivation might be affected by these changes and which parts are definitely not affected by these changes.

When given a history *node* to *mark* as being possibly affected by a change, the routine in the implementation that does this basically *marks* that *node* as possibly being affected and then goes on to recursively *mark* in a similar fashion

1. the node(s) corresponding to expressions that contain this *node*
2. the node(s) that were created (directly or indirectly) as a result of transforming this *node* (recurse down the *xformed-into* attributes, and also the sub-expression links of the nodes which were the values of those attributes)
3. the node(s) that were created (directly or indirectly) as a result of transformations that were *supported* by (depended on) this *node* in some way (recurse down the inverted *support* links, and also the sub-expression links of the nodes which were the values of those inverted links)

3.4 Finding the Derivation Results Unaffected by Changes

Once all the changes to the initial artifact have been made, the incremental rederivation system needs to find the parts of the derivation that were definitely *not* affected by the changes and show what the results are from all those unaffected parts. This process is what saves one from having to replay all the transformations after changes are made to the initial artifact.

Before going into how the current implementation uses the history to determine what the results are from unaffected parts of the derivation, the following should be mentioned: the part of the implementation that actually makes the changes to the initial artifact has yet to be written. The implementation assumes that those changes are also made to the derivation history, so that after the changes are made, the derivation history will be in an "in-between" state: it will reflect the new initial artifact, but still also reflect the (now partially invalid) original set of transformational derivations applied to the original initial artifact.

To determine the results of the unaffected parts of the derivation, the implementation performs the following procedure on the top node in the history (call it α), which corresponds to the top level expression in the now changed initial artifact:

[8] The *supporting* nodes also store what transformations might have needed them.

- Examine α for the following:

  1. α is *not marked* as possibly affected by the changes,
  2. α has been transformed at some point (call the resulting node β), and
  3. β is also *not marked* as possibly affected by the changes.

- If all 3 conditions are satisfied, α has been transformed and the transformation is known *not* to be affected by the changes made to the original artifact. The transformation result β is now of more interest than α, so the implementation will restart this procedure with β in place of α via a recursive call and return the results of this recursive call.

- If it is *not* the case that all 3 conditions are satisfied, then either α has not been transformed, or the transformation has been possibly affected by the changes to the initial artifact. Now the implementation will make a copy of α, replace the subexpression nodes in that copy with the results of recursive calls of this procedure on those subexpression nodes, link the copy to the original node (the copy is considered the "current artifact" version of the original node, which is a history node) and return the now altered copy.

The result of calling this procedure on the top history node is an AST representing the changed initial artifact after it has been transformed by all the transformations known *not* to be affected by the changes to the initial artifact. This AST is now considered the current artifact. The tendency of this procedure is to recurse down the subexpressions of the initial artifact copying those subexpressions' nodes to the extent the procedure is not diverted by some transformation that is known *not* to be affected by changes to the original artifact (in other words, known to have the same results both before and after the changes to the original artifact).

3.5 Finding and Replaying the Possibly Affected Transforms

After the effects of the unaffected transformations have been accounted for, the transformations that may have been affected by the changes need to be found and replayed in the relative order in which they were originally run. At the end of replay, the current artifact and history AST's will have been updated with the replayable transformations, and a list of the unreplayable transformations (if any) will exist. For the most part, finding such transformations just involves searching through the history AST to look for all transformations that have been *marked* as possibly affected by the changes.

A complication of this search process occurs when part (call it P) of the result of one possibly affected transformation (call it α) is used as part of the input to another transformation (call it β).[9] The complication is because at the time β is replayed, α will have already been replayed, so P, which β needs, may no longer be there.

The way that the implementation handles this complication is that when replaying β, the implementation substitutes P with whatever positionally corresponds to P in the result of replaying α. So for example, if P is the 2nd element of the 3rd element in the result

[9] β will also be possibly affected because if nothing else, it is dependent on α, which is possibly affected.

from the original use of α, then when replaying β, substitute P with the 2nd element of the 3rd element of the result from replaying α. If this part of the result cannot be obtained for some reason (e.g., α cannot be rerun or the result of rerunning α does not have at least 3 elements), then β is labeled as unreplayable.

4 Related Work

Concentrating on rederivation in maintenance applications is an interesting simplification to adopt since it finesses the correspondence problem that occurs in other reuse strategies such as applications of derivational analogy. Adopting a dependency-directed impact analysis approach obtains the benefits of typical serial replay approaches without incurring the overhead of manual requerying and dead-end derivation adaptation while trading the serial replay time cost for a memory cost.

The replay work in the KIDS system [5] emphasizes a derivational analogy approach [9] to reuse concentrating on the correspondence problem. This approach is particularly useful in reusing a previous solution to solve a new but similar (analogical) problem. In our work, we are assuming the problem is more than similar but in fact substantially identical except for a defined delta corresponding to a maintenance update.

Baxter takes an approach to reusing lengthy design histories [4] that involves propagating maintenance deltas through the design history to obtain a reordering of the history into a prefix sequence that is unaffected by the maintenance delta (or at least reusable) and the balance of the history that is no longer appropriate. The reusable prefix is then rerun and problem solving proceeds from that point. This is done taking the goal structure of the design history into account. We take a maintenance delta approach, like Baxter, but propagate a delta through a transformation dependency structure instead of the design history. PRIAR [10, 11] also takes into account the dependency structure of a derivation, in this case a plan, to find a maximally reusable plan. As in Baxter's work, the unnecessary parts of a reused plan are removed and new parts are added to establish unmet goals. Other approaches for reusing maximal parts of a design history and then reinvoking a performance program include the REMAID [12] experiments.

Feather's work on parallel elaboration via evolution transformations and merging [13, 14, 15] points to a transformational development methodology that should yield highly replayable derivation histories. Elaboration of parallel design considerations allows non-interfering parts of a development to emerge separately. Where the parallel paths need to be merged, the merge process will identify dependencies. Feather assumes a serial replay process as one mechanism to achieve merging. Our dependency-directed approach does not seem initially applicable to achieving merge via replay since the nature of the merge process is to resolve undocumented dependencies. However, the dependency-driven approach should be highly effective on the resulting replayable derivation histories.

5 Status and Conclusions

We have defined the requirements for our replay capability, have designed the supporting representations and some of the algorithms, and will be developing the protocol of interac-

tion for design tools desiring to use this replay service. Most of the implementation for the macro-level rederivations has been written and tested for "correctness". The exception is (as mentioned in subsection 3.4) that the part that actually makes alterations to the initial artifact has yet to be written. The portion of the implementation for the micro-level rederivations also has yet to be written.

Beyond completing the implementation, it needs to be tested on some decently sized examples of transformational developments to see how useful the idea of storing all this history information is. We recently attended KBSE '92, where we met some people who may be able to supply us with such examples, including Elaine Kant, Henson Graves, Toru Yamanouchi,[10] Y. Ledru, and Lawrence Markosian. One critical measure of evaluation of this approach is memory overhead, which we will need to monitor closely.

The incremental dependency-directed rederivation approach we have described is an excellent match to maintenance type artifact modifications in which the most difficult part of the problem is identifying impacted areas. Having performed this impact analysis, maintenance can proceed on only those areas affected. Micro-level rederivation strategies will allow us to limit those areas even further. The replay delta calculus approach defines the possible impacts of a maintenance delta.

The KBSA style of program development via rederivation is different than today's code-based maintenance approach. This style will pose new requirements on software development tools, a replay capability being just one of those requirements. Another area to be addressed is a new form of complexity metric beyond, for example, the McCabe metrics [17], to assess program complexity. The replay delta calculus may permit the assessment of the modifiability of a derivation history which could serve as just the metric required, measuring the modifiability of the program via the modifiability of its design.

Acknowledgments

This work has been supported by Rome Labs under the umbrella of the Knowledge-Based Software Assistant project. Thanks to Doug Smith and Tom Pressburger for showing us the KIDS replay facility and commenting on this work.

References

[1] C. Green, et al. Report on a knowledge-based software assistant. Technical Report RADC-TR-83-195, Kestrel Institute, 1983.

[2] S. Rowley, H. Shrobe, and R. Cassels. Joshua: Uniform access to heterogeneous knowledge structures. In *6th National Conference on Artificial Intelligence*, 1987.

[3] S. Kambhampati. A theory of plan modification. In *8th National Conference on Artificial Intelligence*, 1990.

[4] I. Baxter. Transformational maintenance by reuse of design histories. Technical Report TR-90-36, UC Irvine, November 1990.

[10] He might run into problems with international export and/or company regulations if he were to give us anything.

[5] A. Goldberg. Reusing software developments. In *4th Annual Knowledge-Based Software Assistant Conference*, 1989.

[6] J. Mostow. Design by derivational analogy: Issues in the automated replay of design plans. *AI Journal*, 40(1-3), September 1989.

[7] R. Balzer. Transformational implementation: An example. *IEEE Transactions on Software Engineering*, 7(1), 1981.

[8] J. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5), 1984.

[9] S. Bhansali and M. Harandi. The role of derivational analogy in reusing program design. In *5th Annual Knowledge-Based Software Assistant Conference*, 1990.

[10] S. Kambhampati. Mapping and retrieval during plan reuse: A validation structure based approach. In *8th National Conference on Artificial Intelligence*, 1990.

[11] S. Kambhampati and J. Hendler. Control of refitting during plan reuse. In *11th International Joint Conference on Artificial Intelligence*, 1989.

[12] B. Blumenthal. Empirical comparisions of some design replay algorithms. In *8th National Conference on Artificial Intelligence*, pages 902–7, 1990.

[13] M. Feather. Specification evolution and program (re)transformation. In *5th Annual Knowledge-Based Software Assistant Conference*, 1990.

[14] M. Feather. Detecting interference when merging specification evolutions. In *5th International Workshop on Software Specification and Design*, 1989.

[15] M. Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, 15(2), 1989.

[16] Reasoning Systems, Inc., 3260 Hillview Ave., Palo Alto, CA 94304. REFINE *User's Guide*, 1990. For REFINE^{TM} Version 3.0.

[17] G. Gill and C. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12), December 1991.

•U.S. GOVERNMENT PRINTING OFFICE [illegible] - [illegible]-60288

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.